ROB313 Assignment 4

Ethan Rajah
1006722370

I. Objectives

The primary objective of this assignment is to explore the implementations of deep neural networks and to make use of the Python libraries that can simplify the process of training the model, both in complexity and runtime. This assignment focuses on multi class classification problems, particularly through the use of the MNIST dataset. The significance of the way in which one initializes the weights is explored within this assignment, as we see that it can enhance the effectiveness of stochastic gradient descent backpropagation. Specifically, one of the common initialization methods for neural networks, Xavier initialization is derived and implemented within the assignment. Furthermore, the computational consequences of neural network implementations are explored through the analysis of the LogSumExp trick and the library functions that address it. Once the neural network is implemented, we study the process of tuning the network to maximize the validation and test accuracies of the 10-class classification problem. Specifically, the methodology behind choosing hyperparameters such as epoch number, learning rate, and the number of hidden layers is explored and analyzed within this report. The convergence trends of the training and validation loss over epoch number is assessed to better understand some of the key limitations and considerations that need to be made when training a neural network. With the completed model, we explore the use of confusion matrices as a way of measuring the precision and accuracy of our model graphically. The advantages of using confusion matrices within the model design process is discussed and future improvements to the model performance are suggested based on the insights received from this plot.

II. Code Structure and Strategies

Most of the neural network implementation was provided in the starter code for this assignment. This section of the report will highlight the additions made to the code to ensure proper functionality and performance of the neural network.

Question 1:

Question 1 required a derivation of Xavier initialization, followed by an implementation of the initialization in code. Since our weight matrix contains the weights for each of the feature/class functions within the 10-class classification problem, we want to initialize it to be of shape M by N, where M is the number of features and N is the number of data points within the set. For this reason, we create a function called ‘init_xavier’ which takes in M and N to specify the weight matrix shape. Another input to this function is an instance of numpy.random’s ‘RandomState’ function, which is a container for the Mersenne

Twister pseudo-random number generator. This random number generator is useful because it has a statistically uniform distribution of values and has a long period, meaning that it can generate a very large number of values before repeating itself. This random state instance is used within the function to generate the M by N weight matrix initialization, which is then divided by the square root of N, the number of input data to complete the Xavier initialization. The reason for this division will become prevalent through the derivation of the initialization in the coming sections of the report.

Question 3:

Question 3 required an implementation of the log-likelihood for multi-class classification. We use the categorical distribution, which is a generalized Bernoulli distribution for K categories, to form the log likelihood for this system. Essentially, we train K separate models to generate K class-conditional probability functions, which denotes the probability of a particular class being true given the data, x. We can represent this in the following form:

$$Pr(y^i|x^i, w) = \prod_{j=1}^{K} \hat{f}_{j=1}(x^i; w)^{y_j^i}$$

Using the categorical likelihood form presented above, we can assume that each data point, x^i, is i.i.d, which allows us to take the product of the likelihood over all N points within the set. With this, the likelihood can be converted to a log likelihood by applying the logarithm to the function. This converts the product operations, over both N and K, to summations. This process is seen as follows:

$$Pr(y^i|x^i, w) = \prod_{i=1}^{N} \prod_{j=1}^{K} \hat{f}_{j=1}(x^i; w)^{y_j^i}$$

$$logPr(y^i|x^i, w) = \sum_{i=1}^{N} \sum_{j=1}^{K} y_j^i log(\hat{f}_{j=1}(x^i; w))$$

This gives us the dataset log likelihood for a K-classification problem. To implement this within a function called 'mean_log_like', we recognize that since we use a weight matrix (10xN) to represent the weights for classifying all 10 classes, our log likelihood function does not require a summation over K probability functions. Since our function 'neural_net_predict' already computes the normalized class log probabilities for our 10 class classification function $\hat{f}$ using the weight matrix, we call upon it in this function and multiply its output by the target classification results, which would be y_train, y_valid, or y_test, when running the code. This allows us to compare the model classification likelihood to the true results to assess how well the model is performing over the epoch number. This

result is an Nx1 vector of log likelihoods, so we sum over the probabilities and divide by the number of input data points, N, to get a mean log likelihood for the model.

Question 4:

Question 4 focused on the tuning of hyperparameters to achieve a final model that had a validation accuracy of at least 95%. Once this was complete, a dictionary was used to store the loss in accuracy using the accuracy results computed within the 'accuracy' function. The dictionary, which was already storing the negative log likelihood measurements for each epoch, is then used to produce a plot for the training and validation loss over epoch number. Moreover, using the final model, the test set was used to illustrate the test accuracy over the epoch number.

Question 5:

Question 5 suggested an implementation of the confusion matrix for the final model to better assess the accuracy and precision of the model when applied to the validation set. To do this, sklearn's metrics library was used as it provided two useful functions, 'confusion_matrix' and 'ConfusionMatrixDisplay'. The 'confusion_matrix' function takes the target classification data and the model predictions as inputs to compute the confusion matrix, whereas 'ConfusionMatrixDisplay' takes the confusion matrix results and creates a visualization instance of matrix with a heat map coloring to show how frequent correct classifications were done by the model, vs. the number of misclassifications. These library functions were implemented within 'getConfusionMatrix', which is a function that is called after the Adam optimization, a variant of SGD that makes better use of gradient information, is completed. This function takes in the validation set model predictions and its corresponding target classification data as input. To retrieve the final predictions of the model at the last epoch of the optimization process, the dictionary used in question 4 is modified to include a predictions list to store the final results from predicting on the validation set. This is then sent to 'getConfusionMatrix' along with the target results to create and plot the confusion matrix. To note, the target results were passed into the function as a Nx1 array, rather than an NxD array due to the constraints of sklearn's 'confusion_matrix' function. It cannot handle multi-dimensional target input, so instead, the indices of the true classification for each data point within the validation set were passed in as the targets. This was done by computing the argmax of the validation targets over axis 1, as through one-hot encoding, the true class for each data point has a value of 1 and the rest are 0. With this adjustment, the confusion matrix was able to be calculated and plotted using sklearn's libraries.

III. Results and Discussion

Question 1 Derivation:

We are given a hidden unit output in the following form, $z_i = \sigma(\sum_{j=1}^{D} w_{ij}x_j)$, where $x_i \sim N(0, \eta^2)$, $w_{ij} \sim N(0, \epsilon^2)$ and σ is a linear (identity) activation function. Furthermore, since we want to derive Xavier initialization, we particularly want to show that the variance of the output of any hidden layer is the same as the variance of the input data. This means that we want to derive a variance for the weights that preserves the variance inherent in the data for each hidden layer. Since x_i and w_{ij} are independent random variables, the derivation is as follows:

$$Var(z_i) = Var(\sum_{j=1}^{D} w_{ij}x_j)$$

$$Var(z_i) = Var(\sum_{j=1}^{D} w_{ij})Var(x_j) + Var(\sum_{j=1}^{D} w_{ij})E[x_j]^2 + Var(x_j)E[\sum_{j=1}^{D} w_{ij}]^2$$

$$Var(z_i) = Var(\sum_{j=1}^{D} w_{ij}) \times Var(x_j)$$

$$\eta^2 = D\epsilon^2\eta^2$$

$$\epsilon = \frac{1}{\sqrt{D}}$$

This result gives us that the variance of the weights should be defined as the inverse of the square root of the number of inputs to the neuron. Note that in this derivation, we make use of the fact that the expectation of both x_j and w_{ij} is 0, hence, two of the terms in line 2 of the derivation are disregarded. This result, as previously mentioned, was used in the code implementation of Xavier initialization by dividing each element of the randomized weight matrix by the square root of the number of input data points to the layer. Recall that we use initialization techniques such as Xavier initialization to prevent activation functions from causing all of the gradients to be zero when completing gradient descent. This means that using initialization techniques helps to prevent gradients from exploding or vanishing too quickly, which would affect the performance of the final model. We also do this to aid in improving the conditioning of the optimization landscape, similar to how we normalize training inputs to zero mean and unit variance. Resultantly, the use of initialization for the weights of the model significantly improves the accuracy of the neural network, thus requiring less computational time to converge to a satisfactory model accuracy.

Question 2:

Upon analysis of the 'neural_net_predict' function provided, it is clear that the use of the logsumexp function to compute the log probabilities is used instead of a more naive approach such as the explicit form seen in the assignment because of the risk of the logarithmic sum of exponents of the outputs blowing up during computation. This issue has been addressed through the LogSumExp trick, which provides an alternative form to doing this computation, as proved in lecture. Essentially, this trick shifts the center of the exponential sum using a regulation term, 'a', which is set to be the maximum output value, so that any unstable output value does not result in the computation blowing up. The logsumexp function from scipy takes this alternative approach to compute the log sum of exponentials of the outputs, thus resulting in a more stable computation for the program. This function helps us to prevent underflow and overflow errors, which would cause catastrophic errors within the training process of the neural network, making the resulting model, if even produced, very unstable and inaccurate. Furthermore, the naive approach could crash in multiple instances due to division by 0 or when dealing with inf/-inf values produced in the program (i.e. from limited precision of the computer), therefore causing numerical instability. Overall, the LogSumExp trick as implemented by the logsumexp function prevents these issues from occurring, which ensures that our program is robust when dealing with any form of output model results in the neural network.

Question 4:

Epoch	Train Accuracy	Validation Accuracy	Test Accuracy
0	0.1076	0.096	0.111
1	0.9204	0.905	0.931
2	0.9548	0.926	0.949
3	0.9736	0.94	0.948
4	0.9798	0.943	0.954
5	0.9879	0.946	0.956
6	0.9918	0.945	0.96
7	0.9918	0.953	0.958
8	0.9933	0.945	0.962
9	0.9885	0.941	0.957

10	0.9942	0.946	0.961
11	0.9943	0.953	0.957
12	0.9924	0.945	0.965
13	0.9944	0.948	0.962
14	0.9972	0.951	0.962

Table 1: Final neural network results with tuned hyperparameters

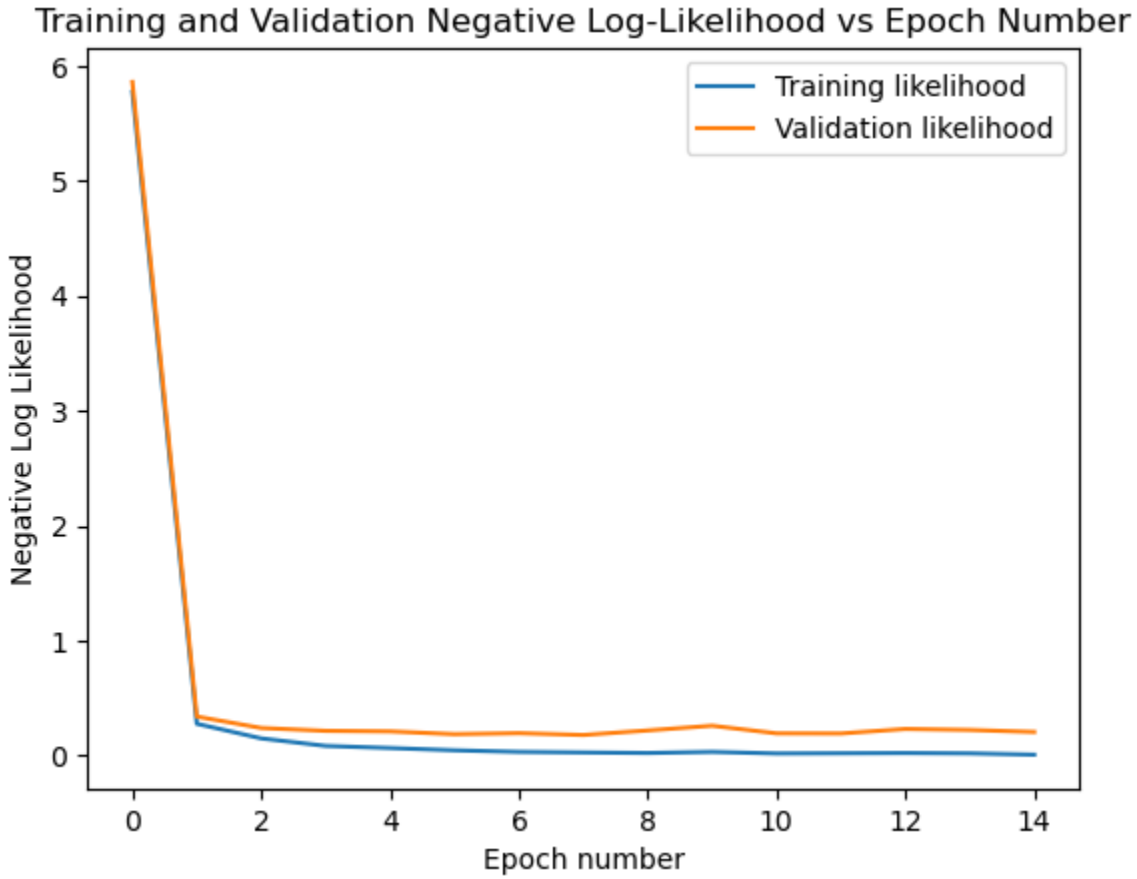

Figure 1: Negative log-likelihood over epoch number for training and validation sets

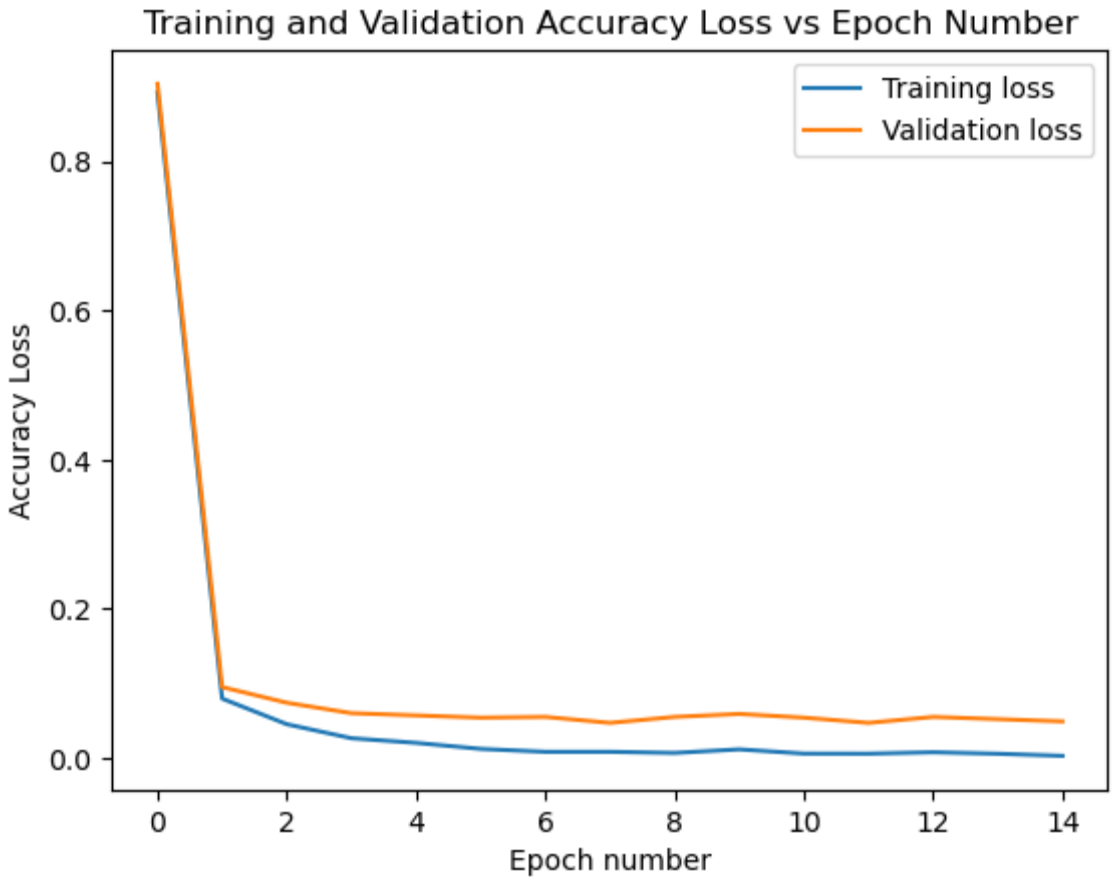

Figure 2: Training and validation set accuracy loss over epoch number

Table 1 shows the training, validation and test accuracy results of the final tuned model. My goal was to maximize the accuracy of these sets, while ensuring that computation time was not compromised. With this goal in mind, I found that 600 hidden layers, 15 epochs and a learning rate of 0.01 provided the best results for both accuracy over 95% for the validation and test sets, as well as in terms of computational time. We see from this table that we reach an accuracy over 95% for all three sets prior to epoch 15, however, these results become more stable to an accuracy of 95% as we approach the 15 epochs. The issue of the stability of convergence is seen within **Figure 1** and **Figure 2**, as the loss has some small oscillatory behavior when converging to the minima. The plots show that this behavior is smoothened out as we increase the epoch number, which is why I decided on an epoch limit that was greater than 7 (the first instance where the validation accuracy was seen to exceed 95% as shown in **Table 1**). These figures maintain the same shape, which is expected between the accuracy and negative log likelihood representations of loss. The significance of these two plots, aside from showing the small oscillatory behavior of the loss over epoch number, is that they show that the validation loss never completely converges to the training loss. This illustrates that there are improvements that

can be made to the model to achieve higher accuracies and thus a greater loss convergence between the training and validation losses. However, based on our design constraint that indicates that 95% validation accuracy is good enough for our model, the difference in convergence between the training and validation sets can be ignored.

When tuning the hyperparameters for the model, I found that increasing the number of hidden layers had a large effect on improving the accuracy, however, as the number of layers increased to around 1000, the accuracy had little to no improvement from a version of the model that had 600 layers. In this case, I was not adjusting any of the other hyperparameters. This is likely why it is recommended to complement an increase in the number of hidden layers with an appropriate learning rate, otherwise we involve additional computations that serve no purpose aside from increasing the runtime of the training process. I also observed that when decreasing the learning rate, I needed to increase the epoch number because otherwise, it would not converge to an accurate enough model when comparing it to using a larger learning rate with the same epoch number. This was an expected observation as we saw in assignment 3 that with smaller learning rates, we have slower convergence and thus a greater number of epochs is needed to allow for the slower convergence to occur.

Using this information, my tuning procedure began with testing how high of a validation accuracy I could get by maintaining a learning rate of 0.1 and increasing the number of hidden layers by 50 each time. After increasing the number of hidden layers past 700, I noticed the accuracy became stationary at around 93% validation accuracy with increasing runtime. For this reason, I increased the number of epochs from 7 to 10 and capped my number of hidden layers to 700, which resulted in an accuracy of about 94.5%. Since this was not enough to suit the design requirements for the model, I decreased the learning rate to 0.01 and kept the rest of the hyperparameters the same. This got me to a maximum of 95.4% accuracy, however, this was not stable as with the next iterations, the accuracy would drop back down to 94%. For this reason, I increased my epoch limit from 10 to 15 to allow for the model to converge away from the small oscillatory behavior of the loss, as previously mentioned. With these results, the design constraint for the validation accuracy was satisfied, however, I wanted to attempt to decrease the runtime if possible. For this reason, I tested decreasing the number of hidden layers from 700 at 50 layer decrements to assess when the validation accuracy would drop below 95%. From this, I observed that for my selected hyperparameters, anything under 600 would result in the validation accuracy being unable to converge to 95% over the 15 epochs. Thus, I chose 600 as the number of hidden layers as it reduced the runtime of the program by a ~3s and was sufficient for reaching a 95% accuracy by the final epoch. This stability of model accuracy is further shown by the test accuracy in **Table 1** as there appears to be little variance between

the accuracies as we approach the final epochs. Moreover, the validity of my tuning process was supported by a test accuracy of 96%, which is above the design goal for the model.

Question 5:

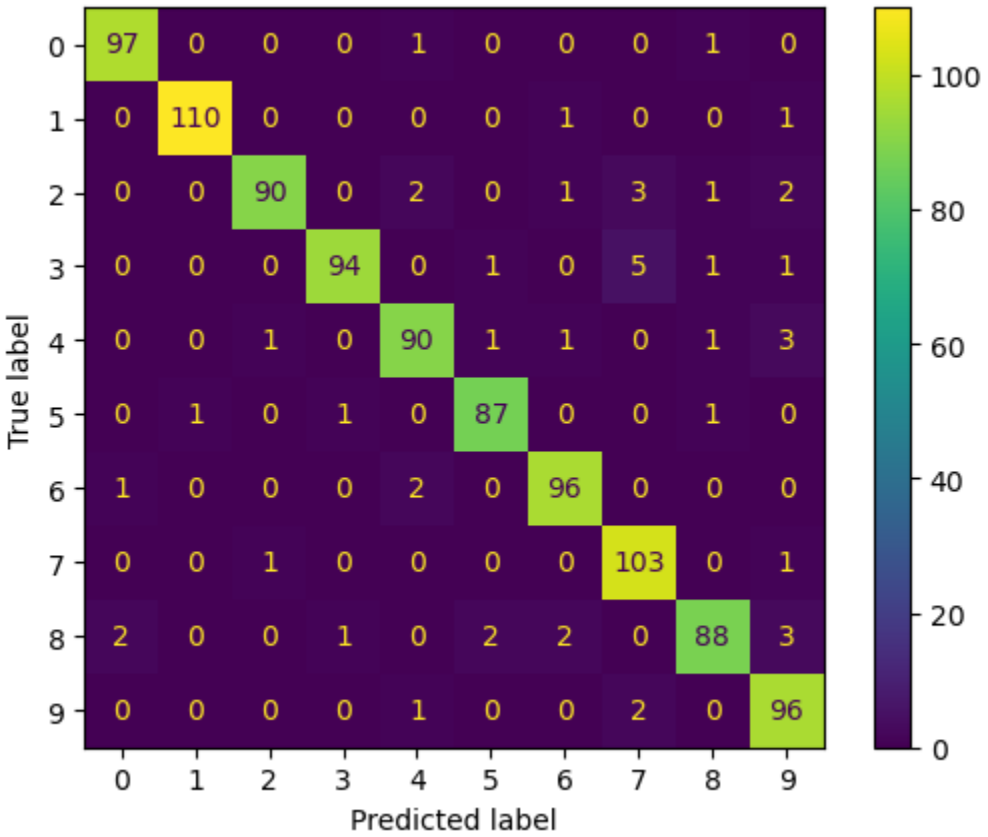

Figure 3: Visualization of the confusion matrix for the final neural network model

This question provided an introduction to confusion matrices and their application as a more detailed performance metric for a neural network. A confusion matrix analyzes the potential of the classifier model; it provides you with a better idea of the types of errors that the model is making during the classification process. In **Figure 3**, the diagonal elements denote the correctly classified outcomes, whereas the misclassified outcomes are indicated everywhere else with a number supporting the number of times the misclassification occurred. This means that the best classifier model would have a confusion matrix with full diagonal elements, where the rest of the matrix elements are set to 0. Essentially, the confusion matrix shows what prediction labels the model got confused with when trying to match the true label and identifies the number of times it occurred.

In the confusion matrix for our model, as shown in **Figure 3**, we see that the network confuses the true label of '3' with '7' the most in the MNIST dataset, with a total of

5 mistakes. Aside from this, we most commonly mistake numbers such as ‘4’ and ‘8’ with the number ‘9’ as our prediction of 9 (last column of the confusion matrix) has the most inaccurate classifications of the digits available to classify with. This is significant because it indicates that one strategy to further improve our model would be to provide more data on the numbers ‘7’ and ‘9’ as we see that the current model misclassifies the true label using these digits most often. Moreover, more data on the numbers ‘3’, ‘4’ and ‘8’ would also be useful for generating a more accurate model as it will help the network in ensuring it does not confuse handwritten variations of those numbers with ‘7’ and ‘9’ as previously mentioned. Another strategy to further improve our model would be to use the confusion matrix to optimize the network for precision. When optimizing for precision, we are attempting to reduce the variance of false classifications among the digits that the model can decide upon. Essentially, we want to improve the model by adjusting hyperparameters such that if it does make false classifications, those false classifications are concentrated towards the same digit, rather than having them spread out among many digits as we see with **Figure 3**. This will allow us to narrow down on which digits require more data to remove the confusion in the model’s performance, rather than providing large amounts of additional data for multiple digits, which will worsen runtime and may be challenging to obtain if applying a similarly structured neural network to a more abstract dataset. Overall, these strategies, both precision optimization and additional data feeding can be used together to generate a more accurate model that has minimal effects on runtime.